

# **Recommandations**

## **Projet plateforme de cryptomonnaie société GitMeMoney**



**GITMEMONY**

## Auteur(s) et contributeur(s)

Nom & Coordonnées	Qualité & Rôle	Société
Gérald ATTARD	Architecte logiciel	GitMeMony

## Historique des modifications et des révisions

N° version	Date	Description et circonstance de la modification	Auteur
1.0	12/12/2022	Création du document	Gérald ATTARD

## Validation

N° version	Nom & Qualité	Date & Signature	Commentaires & Réserves
1.0	Natalia Team leader		

## Tableau des abréviations

<b>Abr.</b>	<b>Sémantique</b>
<b>ARC</b>	<b>Architecte</b>
<b>CTO</b>	<b>Chief Technology Officer</b>
<b>Daily</b>	<b>Daily Meeting</b>
<b>DevTeam</b>	<b>Equipe de Développement</b>
<b>DpC</b>	<b>Déploiement Continue</b>
<b>DOD</b>	<b>Definition Of Done</b>
<b>DOR</b>	<b>Definition Of Ready</b>
<b>ER</b>	<b>Equipe de Réalisation</b>
<b>IC</b>	<b>Intégration Continue</b>
<b>LC</b>	<b>Livraison Continue</b>
<b>PM</b>	<b>Product Manager</b>
<b>NR</b>	<b>Non Régression</b>
<b>PB</b>	<b>Product Backlog</b>
<b>PO</b>	<b>Product Owner</b>
<b>QA</b>	<b>Quality Assurance</b>
<b>SB</b>	<b>Sprint Backlog</b>
<b>TDD</b>	<b>Test Driven Development</b>
<b>UML</b>	<b>Unified Modeling Language</b>
<b>US</b>	<b>User Story</b>
<b>CTO</b>	<b>Chief Executive Officer</b>
<b>PMAQ</b>	<b>Plan Management et Assurance Qualité</b>

## Table des matières

I. Préambule.....	5
I.A. Objectif du document.....	5
I.A.1. Définition de l'Intégration Continue :.....	6
I.B. Description du besoin.....	6
I.C. Investissement à fournir.....	7
I.D. Bonnes pratiques.....	8
II. Pratiques collaboratives.....	9
II.A. La métaphore.....	10
II.B. La programmation en binôme.....	11
II.C. La responsabilité collective du code.....	11
II.D. Les règles de codage.....	13
II.E. L'intégration continue.....	13
II.E.1. Rôle des tests unitaires.....	14
III. Composants Agiles.....	15
III.A. Les rôles.....	15
III.A.1. Product Owner.....	15
III.A.2. Scrum Master.....	16
III.A.3. Equipe de développement.....	16
III.B. Les artefacts.....	17
III.B.1. Definition Of Ready.....	17
III.B.2. Définition Of Done.....	18
III.B.3. Epic.....	19
III.B.4. User Story.....	19
III.C. Les jalons et les cérémonies.....	20
III.C.1. Sprint.....	20
III.C.1.a. Préparation du <i>sprint</i> .....	20
III.C.1.b. Périmètre et planification d'un <i>sprint</i> .....	20
III.C.1.c. Méthode de travail et déroulement d'un <i>sprint</i> .....	21
III.C.1.d. Conclusion d'un <i>sprint</i> .....	22
III.C.1.e. Le <i>Sprint0</i> , un <i>sprint</i> spécifique.....	22
III.C.2. Cérémonies.....	23
III.C.2.a. Sprint planning.....	23
III.C.2.b. Sprint review.....	23
III.C.2.c. Sprint retrospective.....	24
III.C.2.d. Daily meeting.....	25
III.C.2.e. Sprint refinement.....	25
III.D. Outils organisationnels.....	26
III.D.1. Product Backlog.....	26
III.D.2. Sprint Backlog.....	27
III.D.3. Release Plan.....	28
III.D.4. Vitesse et point d'US.....	28
III.D.4.a. Calcul.....	29
III.D.5. <i>Poker Planning</i> .....	30
III.D.6. Graphique de <i>sprint burndown</i> .....	31
III.D.7. Graphique de <i>sprint burnup</i> .....	32
IV. synthèse.....	33



# I. Préambule

## I.A. Objectif du document

Ce document a pour objectif de fournir des préconisations relatives à la création d'une plateforme d'interconnexion de différents sites marchands de cryptomonnaie à l'initiative de la société *GitMeMoney*. Ainsi, ce document proposera un ensemble de pratiques, consécutives à la production de code source, consistant à :

- vérifier les activités et tâches fonctionnelles définies à chaque jalon temporel ;
- vérifier que le résultat de ces modifications et ajouts n'entraînent pas de régression fonctionnelle ;
- livrer le résultat obtenu en utilisant un emplacement fiable, sécurisé et pérenne.

Cet ensemble de pratiques est qualifié d'**Intégration Continue** (IC) pour laquelle chaque brique logicielle est à spécifier dans des procédures techniques spécifiques à chacune d'entre elle, et indépendantes les unes des autres.

Ces briques logicielles prises dans leur ensemble devront aboutir sur la mise en place d'un système d'automatisation des tâches induites en Génie Logiciel, à savoir : compilation, tests unitaires et fonctionnels, validation produit, diverses analyses du produit (test de charge, de rendement...), livraison continue ; les tâches incombant à la maintenance n'étant pas définies dans la portée de ces recommandations.

Toutes ces briques logicielles vont se voir également intégrées dans une démarche globale d'intégration AGILE, en vue d'obtenir, pour ce projet, de :

- l'**Adaptation** : les ajustements doivent être rapides et proportionnés afin de limiter les dérives (adaptation des ressources, des outils, des processus...);
- la **Transparence** : la définition d'un vocabulaire commun, idoine au projet, sera nécessaire pour l'ensemble des intervenants, y compris les parties prenantes, afin que tous aient une compréhension commune de ce qui a été produit et donc montré ;
- l'**Introspection** : l'équipe projet sera consciente des compétences à forte valeur ajoutée à obtenir pour le bien du projet, et connaîtra leurs points faibles pour pouvoir y pallier. Le mot d'ordre ici est **Amélioration Continue**.

Comme la question de Qualité logicielle est ici une finalité, il est nécessaire d'en définir la portée. Or ce document n'en a pas la vocation et le lecteur est invité à faire ses propres recherches pour définir cette portée. Néanmoins, toutes les pratiques proposées dans ce document sont inspirées du *Lean Management*, des approches dites *AGILES* et des principes d'*Edwards Deming*, considéré comme le père de la Qualité au sein du contexte industriel.

### **I.A.1. Définition de l'Intégration Continue :**

*"L'intégration continue est un ensemble de pratiques utilisées en génie logiciel consistant à vérifier à chaque modification de code source que le résultat des modifications ne produit pas de régression dans l'application développée. Le concept a pour la première fois été mentionné par Grady Booch et se réfère généralement à la pratique de l'extreme programming. Le principal but de cette pratique est de détecter les problèmes d'intégration au plus tôt lors du développement. De plus, elle permet d'automatiser l'exécution des suites de tests et de voir l'évolution du développement du logiciel."*

Source : *Techniques d'amélioration continue en production* de Robert CHAPEAUCOU aux éditions DUNOD.

## **I.B. Description du besoin**

Les exigences de qualité inhérentes à des travaux de développement en équipe, la diversification des projets, ainsi que le renouvellement des participants, conduisent *GitMeMoney* à utiliser des outils de gestion de configuration, tels que *svn*, *cvs* ou *git*, et de développement dirigé par les tests (*Test Driven Development*).

Ainsi, l'objectif de ce document s'inscrit dans la volonté de *GitMeMoney* de mettre en œuvre des pratiques de développement informatique visant un haut niveau de qualité au sein de ce projet.

C'est dans ce contexte que l'utilisation de l'IC entre en jeu, afin d'avoir un regard constant sur l'état des développements, ainsi que sur la qualité du code fourni.

En effet, le but est de détecter au plus vite l'ajout d'éventuels problèmes ou erreurs au code existant, ainsi que de contrôler la qualité du code. Cela passe par la détection d'erreur, la vérification des résultats des tests, la vérification de leur couverture, ou encore la détection de code dupliqué.

La mission principale consiste en la mise en place d'un système d'IC, équipé d'outils de contrôle de Qualité (couverture par les tests, contrôle de la redondance de code, etc.) au sein de *GitMeMoney*.

Le système proposera des solutions permettant de travailler efficacement dans divers langages, tels que le Java pour le back-end, ou encore le PHP pour le front-end, pour ne citer que quelques possibilités.

Des prototypes d'applications basiques dans ces différents langages, avec des systèmes de construction divers, ainsi que la rédaction de tests unitaires, permettent ainsi de tester et valider la mise en place de la solution.

Un autre objectif de ce document consistera à donner des pistes relatives à la mise en place d'environnements de tests pour les codes précédemment déployés à intégrer en l'état sur ce système d'intégration continue. Des solutions de type machine virtuelle ou conteneur (type *Docker* ou *Jenkins*) seront étudiées et la pertinence de leur couplage au système d'intégration continue sera également à prendre en compte.

## I.C. Investissement à fournir

La première question à se poser sera relative aux différences existantes entre l'intégration classique et l'intégration continue.

En ce qui concerne l'intégration classique, celle-ci consiste à assembler, à la fin du projet, toutes les fonctionnalités développées. Ainsi, suite à la liaison de toutes ces parties logicielles, une longue phase de test et de correction d'anomalies est entamée. Il faut garder toutefois à l'esprit qu'il est extrêmement complexe et chronophage de corriger un code déjà réalisé plutôt que de tester celui-ci au fur et à mesure de sa rédaction. Cette étape entraîne donc des coûts supplémentaires et des délais pouvant impacter fortement la vie du projet.

L'intégration continue (IC) essaie de répondre aux difficultés énoncées ci-dessus. En cela, l'IC va segmenter les différentes étapes par l'utilisation de plusieurs outils spécialisés dans chacun de ces domaines interconnectés. Cet ensemble d'outils permet ainsi de gérer finement l'avancée du projet, de configurer chaque étape de compilation et de générer des rapports ou des métadonnées à propos du projet. Le cycle itératif d'un élément de projet se découpe en 5 principales étapes:

- **Etape 1** : les différents développeurs travaillent de concert pour développer les fonctionnalités et modules qui leur sont attribués. Ainsi, chacun d'eux veille à tester son code à l'aide de tests unitaires et s'assurent de sa bonne intégration au sein du système. Une fois testé chaque développeur fusionne son code avec le code existant. Par la suite, les autres développeurs pourront accéder à cette modification afin de le relire, l'optimiser si nécessaire et le mettre à jour.
- **Etape 2** : l'utilisation d'un outil de gestion de version permet d'avoir une certaine visibilité sur l'avancée du projet. Ce suivi sera alors réalisé progressivement au fur et à mesure des fusions (*merge*) de modifications réalisées par les développeurs.
- **Etape 3** : la compilation de chaque modification de code sera automatiquement associée à l'exécution automatique de tests fonctionnels préalablement écrits en fonction du *Product Backlog* existant.
- **Etape 4** : après la compilation réalisée et les tests exécutés, des informations quant au succès ou à l'échec de ces opérations sont remontées aux parties prenantes. Ces rapports pourront concerner la Qualité, la stabilité ou la découverte d'anomalie non répertoriée jusqu'à présent.
- **Etape 5** : l'analyse de ces rapports va permettre de reprendre le code incriminé et de l'améliorer pour qu'il réponde au besoin pour lequel il avait initialement été rédigé.

Grâce aux tests très réguliers du code rédigé, les anomalies sont détectées dès leur apparition. Il est ainsi plus aisé et plus immédiat de corriger ces anomalies en reprenant une petite partie de code - la seule partie ciblée par le test unitaire. Ainsi, le projet n'étant pas terminé, il y aura, dès le moyen terme, moins de parties de code à modifier, et donc un énorme gain de temps in fine.

## I.D. Bonnes pratiques

La mise en oeuvre de l'ensemble de ces étapes précédentes nécessitera que :

- les tests unitaires soient écrits AVANT la rédaction du code source selon la méthode TDD (*Test Driven Development*);
- un dépôt unique soit identifié et structuré pour permettre un accès au code source à chaque membre de l'équipe (la constitution de cette équipe sera définie dans le document joint à cette étude nommé *Contitution de l'équipe technique*);
- ce dépôt devra également permettre de mettre en place un système et/ou un outil de suivi de version (versionning) dans une optique de traçabilité, de sauvegarde et d'aide au maintien de la vision Produit;
- chaque développeur intègre ses modifications du code source au fur et à mesure du développement de chaque fonctionnalité;
- des tests d'intégration valident, si nécessaire, les imports de version dans une branche du logiciel de suivi de version différente que celle utilisée pour développer la modification du code en question.

Ainsi, les bénéfices, à court terme, de la mise en œuvre de ces "**bonnes pratiques**" seront :

- la vérification fréquente du code source et, donc, de sa bonne compilation;
- la réalisation systématique des tests unitaires et/ou fonctionnels;
- l'alerte immédiate en cas de régression fonctionnelle ou d'incompatibilité de code;
- la détection des problèmes d'intégration;
- la réparation et la maintenance continue, évitant les problèmes de dernière minute;
- au moins toujours une version disponible pour un test, une distribution à fournir ou simplement une démonstration de l'incrément fourni;
- la possibilité d'obtenir périodiquement des remontées d'indicateurs cohérents, relatifs à la Qualité du code, la couverture des tests, la visibilité sur le travail qui a déjà été réalisé et de ce qui reste à faire...





## II. Pratiques collaboratives

Les méthodes Agiles rompent avec l'organisation traditionnelle des équipes de développement selon laquelle les développeurs travaillent seuls sur des tâches et des parties distinctes de l'application. En effet, les méthodes Agiles sont pour l'essentiel basées sur la coopération et la communication : c'est dans la création d'une réelle dynamique d'équipe qu'Agile recherche les performances.

En pratique, l'organisation d'une équipe agile est caractérisée par les deux éléments suivants :

- les membres de l'équipe collaborent et interagissent en permanence ; ils ne travaillent seuls qu'exceptionnellement.
- L'application n'est pas découpée en zones réservées à tel ou tel développeur, mais forme au contraire un tout sur lequel chacun sera habilité à intervenir en fonction des besoins.

Comme ils travaillent toujours ensemble, les membres d'une équipe agile apprennent à se connaître pour opérer ensemble efficacement. L'information circule, les compétences s'échangent ; les problèmes rencontrés lors du développement sont compris puis résolus par l'ensemble de l'équipe.

Chaque membre d'une équipe agile étant amené à travailler sur toutes les parties qui la composent, la connaissance de l'application est mieux distribuée dans l'équipe. L'intervention de plusieurs personnes sur le code (relecture, correction et optimisation) en améliore la qualité et la fiabilité.

En outre, la gestion du projet est d'autant plus facilitée que les développeurs sont capables d'intervenir sur toutes les parties du code de l'application.

Agile tend idéalement à constituer une équipe soudée, flexible, multidisciplinaire et dont la capacité de résolution des problèmes est largement supérieure à celles que peuvent cumuler tous ses membres pris individuellement. Or, cet idéal, presque utopique, est plus facile à décrire qu'à véritablement mettre en place, car l'entente entre les membres d'une équipe ne se commande pas...

Agile fixe cependant cette cohésion comme objectif et propose un ensemble de pratiques concrètes destinées à favoriser son émergence dans le projet. Ce document se propose de décrire 5 de ces pratiques essentielles :

- La métaphore
- La programmation en binôme
- La responsabilité collective du code
- Les règles de codage
- L'intégration continue

## II.A. La métaphore

Au sein des méthodes dites « classiques », la description du projet est réalisée à l'aide de multiples documents issus de sources et de méthodes différentes, telles que la rédaction d'un cahier des charges détaillé, des diagrammes d'architecture plus ou moins complexes, ou encore des documents de conception détaillée en UML... Ces documents peuvent renfermer une quantité considérable d'informations et nécessiter un effort considérable de rédaction non négligeable. Ils jouent alors plusieurs rôles distincts, dont un en particulier : ils permettent à toutes les parties prenantes de se référer à une vision d'ensemble commune du système.

Néanmoins, cette vision d'ensemble est bien souvent noyée par la quantité de documentation nécessaire à sa description, et là où il était question de vision d'ensemble, il ne reste qu'une constellation volumineuse d'informations au sein de laquelle il est maintenant difficile d'appréhender le recul désiré initialement ; autrement dit : « *c'est comme chercher une aiguille dans une motte de foin...* » Ce dicton n'est pas anodin puisqu'il applique lui-même le principe de métaphore.

Avec l'application de méthode Agile, l'équipe va avoir tendance à minimiser les activités qui ne contribuent pas efficacement à l'objectif premier : fournir un logiciel qui fonctionne.

Ainsi, si l'équipe agile a de bonnes raisons de considérer la plus grande partie des documents usuels comme redondante, elle n'en rédigera que les quelques pages dont il n'est pas possible de se passer : celles qui fournissent une représentation d'ensemble.

En général, il s'agira de décrire le logiciel attendu – ou réalisé – en des termes plutôt imagés, évocateurs de sa structure ou de ses parties les plus importantes, et d'éviter les jargons techniques stériles.

A chaque occasion, lorsqu'il s'agira de décrire les fonctionnalités du programme ou sa conception, les membres de l'équipe agile s'efforceront consciemment de trouver des images plutôt que d'en parler de manière technique, comme par exemple : « *cette partie du système fonctionne comme une autoroute, les messages qui circulent dans un sens n'ont pas d'effet sur le traitement des messages qui circulent dans l'autre sens* ».

Cette approche est efficace pour se faire comprendre ponctuellement, même si ces métaphores restent éphémères et n'interviennent plus dans les conversations ultérieures.

Concrètement, la conséquence en est que les membres de l'équipe agile passent moins de temps à discuter de « *ce qu'il est possible de faire dans le système* » et plus de temps à découvrir « *ce que le système doit faire*. »

Ce principe est issu d'un verset du manifeste agile :

~~un anneau pour les gouverner tous~~

**Des solutions opérationnelles, de préférence à une documentation exhaustive**

## II.B. La programmation en binôme

Dans une équipe agile, les développeurs travaillent systématiquement à deux devant une seule et même machine. L'un des développeurs est aux commandes et joue le rôle de **pilote** en prenant à sa charge la rédaction du code lui-même. Il s'occupera donc de manipuler des outils de développement, de la navigation dans le code, des contraintes du langage...en bref, des **aspects tactiques** de la tâche ; ce rôle se rapprochant ainsi de celui d'un développeur solo.

L'autre développeur joue le rôle de **copilote** ; il est chargé des aspects **stratégiques** du développement en cours. Il a pour mission d'effectuer une relecture immédiate et continue du code écrit, de proposer d'autres solutions, d'implémentation ou de design, d'imaginer de nouveaux tests, et de penser aux impacts des modifications sur le reste de l'application, etc. Même s'il n'a pas les commandes, le copilote est loin d'être passif : il est responsable du code écrit au même titre que le pilote. Leur dialogue est donc permanent pour définir les solutions d'implémentation ou de design à mettre en œuvre.

Ces deux rôles ne sont cependant qu'indicatifs. Dans la pratique, pour un observateur extérieur, ces deux développeurs paraîtront simplement entretenir un dialogue permanent, s'affairant à réaliser ensemble la tâche en cours. Le seul élément qui différencie réellement ces deux rôles est donc le contrôle du clavier et celui-ci peut changer de mains à tout moment, les rôles étant alors inversés.

## II.C. La responsabilité collective du code

Par rapport aux modèles « *traditionnels* » au sein desquels la responsabilité du code est individuelle, les méthodes agiles proposent une solution originale sous la forme d'une **responsabilité collective du code**.

Relativement aux paragraphes précédents, chaque binôme peut intervenir sur n'importe quelle partie de l'application dès lors qu'il le juge nécessaire, et chacun est responsable de l'*ensemble* de l'application. En particulier, lorsqu'un binôme découvre qu'une partie de l'application n'est pas aussi claire ou simple qu'elle le devrait. Son rôle va alors consister à améliorer le code en question pour garantir que l'application converge vers un tout cohérent.

Ce mode de responsabilité est parfaitement adapté à la programmation en binôme. Il en reprend tous les avantages relatifs au meilleur partage des connaissances au sein de l'équipe agile, et renforce ceux qui ont trait à la relecture du code, puisque celui-ci est relu non seulement par le copilote et aussi par les autres binômes...

Pris isolément, ce modèle de responsabilité du code peut cependant souffrir de deux principales limitations :

- bien que toute l'équipe agile puisse intervenir sur l'ensemble de l'application, chaque développeur ne connaît parfaitement que le code qu'il a lui-même écrit. Un binôme qui travaille sur une partie qu'il n'a pas écrite risque de passer beaucoup de temps à la comprendre, voire même, d'y introduire des régressions.
- Plusieurs binômes peuvent travailler simultanément sur une même partie de l'application, et la fusion des modifications peut entraîner un surcoût en temps des gestion de configuration du logiciel.

Malgré ces limitations, ce modèle de responsabilité collective se montre parfaitement viable dans un environnement agile au sein duquel ces problèmes sont résolus à l'aide de pratiques complémentaires.

D'une part, l'application est plus facile à comprendre grâce aux effets conjoints de pratiques de conception simples et de remaniement (refactoring), telles que :

- la **programmation darwinienne** : l'utilisation de tests unitaires et le remaniement (refactoring) vont main dans la main (sans tests unitaire, un refactoring de code ne peut être assuré...) ;
- l'**élimination du code dupliqué** : l'un des principaux obstacles à la flexibilité est la présence de fragments de code identiques – ou très similaires – à plusieurs endroits de l'application ;
- la **séparation des idées** : les fonctions ou les méthodes à rallonge sont un autre obstacle majeur à la production d'un code de qualité, souple et évolutif ;
- l'**élimination du code mort** : un code *mort* est une partie du code qui n'est jamais utilisé par n'importe quel autre partie du code de l'application. L'élimination de ce code mort est une des applications du *refactoring*.

D'autre part, les problèmes de modifications simultanées de code sont réduits en raison de l'adoption de démarche d'IC, détaillée dans un paragraphe ci-dessous.

## II.D. Les règles de codage

L'application du modèle de responsabilité collective du code requiert une homogénéisation des styles de codage au sein de l'équipe agile, de sorte que chaque développeur puisse se sentir rapidement à l'aise dans des parties de code qu'il n'a pas écrites lui-même. Pour cela, l'équipe définit au tout début du projet, en général dans le *sprint0*, un ensemble de règles de codage qui sont ensuite scrupuleusement suivies et respectées par chaque développeur.

Ces règles portent sur des aspects divers de la programmation tels que la présentation du code, à l'aide d'indentation par exemple, l'organisation des commentaires ou les règles de nommage des fonctions, variables...

L'introduction de cette pratique présente des risques d'être mal perçue au premier abord car chaque développeur est attaché à son propre style personnel. Néanmoins, l'expérience prouve que l'adaptation à de nouvelles règles et habitudes de codage est « *relativement* » rapide, et que cette pratique permet de faire rapidement prendre conscience à chacun qu'il travaille réellement au sein d'une équipe.

## II.E. L'intégration continue

La pratique de l'intégration continue vise à ce que l'équipe agile soit en permanence en mesure de livrer une version opérationnelle du système – exception faite du *sprint0*. Les avantages d'une telle situation sont nombreux :

- l'équipe évite les problèmes de la « *période d'intégration* » pendant laquelle des efforts disparates sur le système doivent être mis en commun ;
- les efforts de chaque membre de l'équipe agile profitent immédiatement au reste de l'équipe.

Lorsque plusieurs développeurs travaillent sur une même application, il est indispensable que les travaux qu'ils réalisent séparément- individuellement ou en binôme – soient, à un moment ou un autre, répercutés sur une unique version « *officielle* » de l'application : celle qui sera compilée et livrée au client, puis aux utilisateurs finaux.

La nécessité de cette version qui « intègre » toutes les modifications est assez évidente : il est difficile de suivre les évolutions et les dysfonctionnements éventuels d'une seule version d'un programme écrit par un seul développeur ; s'il fallait suivre  $n$  versions différentes par développeur ou par binôme, la tâche deviendrait tout simplement irréalisable.

Cette tâche peut être entièrement manuelle, ou bien elle peut être procédée au moyen d'un outil informatique de gestion de versions.

En tout état de cause, peu importe le choix effectué par l'équipe pour réaliser ce suivi de version, le problème qui se pose reste le même : il s'agit de prendre en compte, ou non, des modifications dans la version officielle.

La réponse à ce problème ne peut venir que d'un seul rôle, celui de *Product Owner*.

## II.E.1. Rôle des tests unitaires

Dans les paragraphes précédents, il a été de nombreuses fois question de test unitaire et ce document se doit d'en donner quelques motivations quant à leur emploi.

En effet, tel qu'il a été écrit précédemment, chaque intégration de code fait habituellement l'objet de tests de non-régression, puisque les modifications séparées des développeurs peuvent avoir des effets inattendus une fois intégrées.

Dans une équipe agile où ces tests sont réalisés ou interprétés manuellement, il n'est tout simplement pas concevable d'intégrer plusieurs fois par jour, puisque cela consommerait beaucoup trop de temps ou ne permettrait pas une couverture de tests suffisante.

Les tests unitaires automatiques préconisés pour ce projet, au sein de *GitMeMoney*, se révèlent donc indispensables pour donner un *feedback* (retour d'expérience) rapide sur le processus d'intégration. Les développeurs peuvent ainsi s'assurer que les tests unitaires passent à 100 % sur leur version locale du code avant intégration, et qu'ils passent toujours à 100 % après l'intégration.

La viabilité de cette pratique dépend de deux facteurs :

- la compilation de l'application et l'exécution de l'ensemble des tests unitaires doivent être très rapides. Cet aspect est primordial en Agile, et l'équipe doit faire le nécessaire sur le plan du matériel, des outils et de l'organisation de son application. Elle doit particulièrement se doter d'un outil de gestion de dépendances efficace qui limite la quantité de code à recompiler après chaque modification.
- Les tests unitaires disponibles doivent être suffisamment couverts pour que l'équipe n'ait pas à se reposer sur les tests de recettes, dont l'exécution est souvent bien plus longue.

*Nota : les tests de non-régression sont destinés à vérifier que les fonctionnalités déjà validées de l'application sont toujours opérationnelles après modification de cette dernière.*



## III. Composants Agiles

Avant de présenter la méthodologie ayant pour vocation à être déployée au sein *GitMeMoney*, il est nécessaire de définir un certain nombre de termes amenés à revenir régulièrement au sein de ce document. Ces définitions sont regroupées selon trois groupes répondant chacun à une question :

- **QUI ?** Cette question définira les acteurs et les rôles appelés à jouer un rôle dans le projet.
- **COMMENT ?** Cette question définira des outils, communément appelés **artéfacts** en AGILE, permettant la mise en place de la méthodologie.
- **QUAND ?** Cette question définira l'ensemble des réunions, appelées **cérémonies** ou **rituels** en AGILE, nécessaires pour définir l'organisation globale du projet, tant en termes de contenu que de positionnement temporel dans la méthodologie de développement du produit.

### III.A. Les rôles

L'ensemble des rôles définis dans les paragraphes qui suivent constituent l'équipe projet.

#### III.A.1. Product Owner

Le Product Owner (PO) est le rôle se rapprochant le plus du client final.

Le PO a la vision fonctionnelle du produit vers laquelle l'équipe de développement doit aboutir, et ne s'occupe que de celle-ci : il n'entre pas dans des considérations techniques, chasse gardée de la l'équipe de développement.

La fonction principale du PO est ainsi de retranscrire les besoins fonctionnels du client au travers d'*epics* et d'*US* (User Story).

Le PO n'est pas seul dans l'élaboration des *epics/US* et est aidé par l'équipe de développement dans le cadre de la cérémonie d'affinage des différentes *epics* du *Product Backlog*.

Le PO est ainsi responsable envers l'équipe de développement de :

- fournir une vision à court et moyen terme du produit, en projetant et en mettant à jour le *Release Plan* ;
- travailler et d'affiner, en collaboration avec l'équipe de développement, les US jusqu'à aboutissement à un DOR (**Definition Of Ready**) ;
- tenir un rythme d'alimentation de l'équipe de développement, c'est à dire d'avoir suffisamment d'US à l'état *ready* (donc avec une *DOR OK*) dans le *Sprint Backlog* ;
- avoir une planification équilibrée ;
- être disponible pour trancher un choix ayant un impact fonctionnel
- définir la forme et le contenu des différents livrables à l'issue de chaque sprint.

### III.A.2. Scrum Master

Le *Scrum Master* est l'animateur Agile. Il est un arbitre maîtrisant le framework Agile mettant l'accent sur des itérations limitées dans le temps appelées *sprints*.

En tant qu'animateur, le *Scrum Master* joue le rôle de *coach* pour le reste de l'équipe. C'est un «*leader-serviteur*», comme le qualifie le *Scrum Guide*.

Ainsi, un *Scrum Master* s'engage à respecter les bases et les valeurs Agiles, et reste flexible et ouvert aux possibilités d'amélioration du workflow de l'équipe.

Le *Scrum Master* aide l'équipe à améliorer et à simplifier les processus par lesquels elle atteint ses objectifs. Il le fait en tant que membre de l'équipe, ou collaborateur, et non pas, idéalement, en tant que contrôleur.

**Les meilleures équipes Agiles sont auto-organisées** et ne réagissent donc pas bien à la gestion descendante.

### III.A.3. Equipe de développement

L'équipe de développement est constituée des personnes qui fabriquent concrètement le produit.

Dans ce projet, l'équipe de développement sera composée de 3 personnes de par les contraintes imposées par les parties prenantes décisionnaires.

De façon générale, une équipe Agile devra être suffisamment grande pour fabriquer le produit et suffisamment petite pour maintenir la flexibilité et l'agilité dans les processus de développement.

L'équipe de développement est une équipe qui s'auto-organise et s'auto-gère pendant chaque *Sprint*.

Elle est une équipe interfonctionnelle, multidisciplinaire et suffisamment qualifiée pour effectuer tout le travail nécessaire dans un *sprint* afin d'accomplir les tâches.

Les principales responsabilités lui incombant sont de:

- travailler et terminer les tâches dans le temps imparti ou les *sprints* ;
- comprendre les exigences fournies par le PO ;
- participer aux cérémonies agiles telles que *daily meeting*, *sprint refinement*, *sprint review*... ;
- suivre les valeurs et les principes de leur méthode/cadre ;
- signaler visiblement la progression de leurs tâches lors du *daily meeting*.



## III.B. Les artefacts

Dans les paragraphes suivants un certains nombre d'outils, nommés **artefacts** en AGILE, seront définis. Ces outils sont des concepts sur lesquels reposent la méthodologie et ont donc une grande importance dans la mise en place de celle-ci et dans son utilisation.


### III.B.1. Definition Of Ready


La *Definition Of Ready (DOR)* est une liste de critères à remplir, définis par le PO et l'équipe de développement, qui cumulés renseignent sur la maturité d'une US.

Si un des critères du DOR n'est pas rempli, alors l'US n'est pas considérée comme « *ready* » et ne peut donc pas être intégrée dans un *Sprint Backlog*.

Si tous les critères du DOR sont remplis, alors l'US est considérée comme « *ready* » : elle devient ainsi intégrable dans un *Sprint Backlog* si le PO le souhaite.

Exemple de DOR :

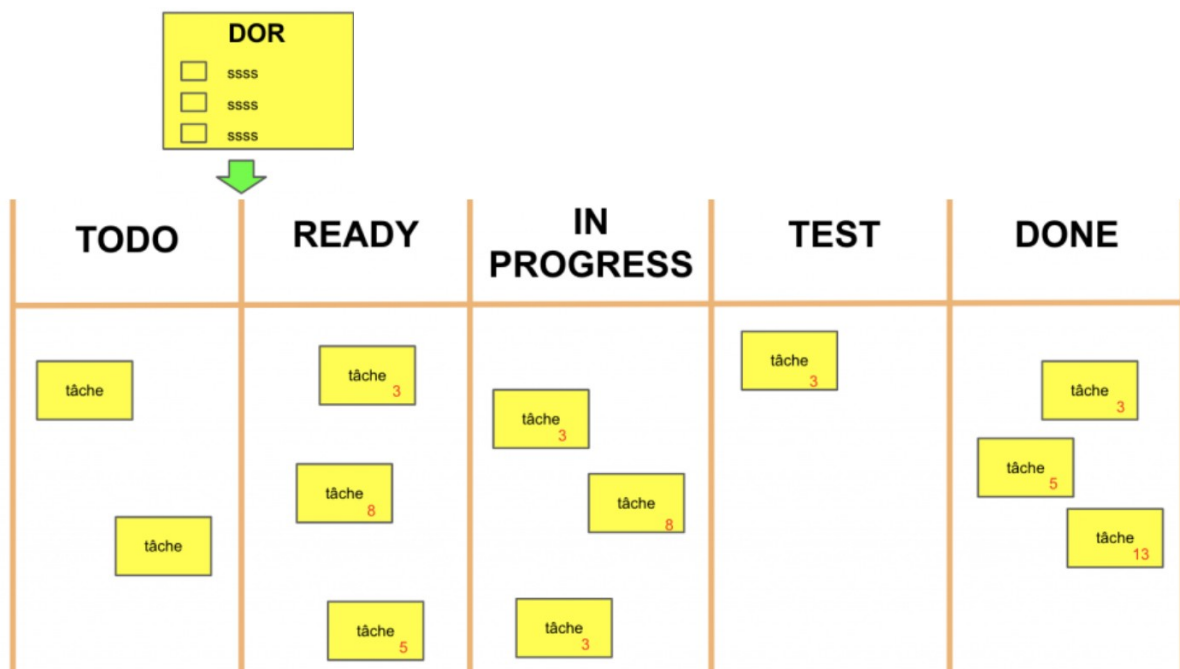
 DOR



**Liste des critères :**

- ☒ L'US possède un libellé clair et compréhensible
- ☒ L'US apporte de la valeur au besoin client
- ☒ L'US est suffisamment découpée
- ☒ L'US est testable et démontrable
- ☐ L'US a tous ses prérequis de disponibles
- ☒ L'US est priorisée
- ☐ L'US est estimée

Cette DOR intervient alors dans le Sprint Backlog comme montré dans le diagramme ci-dessous :



### III.B.2. Définition Of Done

La *Definition Of Done (DOD)* est un artéfact très important pour l'équipe de développement, dans le sens où il définit l'ensemble des critères lui permettant d'affirmer qu'une US est terminée.

La DOD étant une notion très subjective, il est important que tous les membres de l'équipe de développement en partagent la même vision. La DOD doit donc, au même titre que la DOR, être définie dès le début du projet (durant le premier *Sprint* ou *Sprint0*).

Le PO peut être consulté, ou contribuer à l'élaboration de la DOD, mais il est essentiel que cette définition soit rédigée et comprise par l'ensemble des membres de l'équipe de développement.

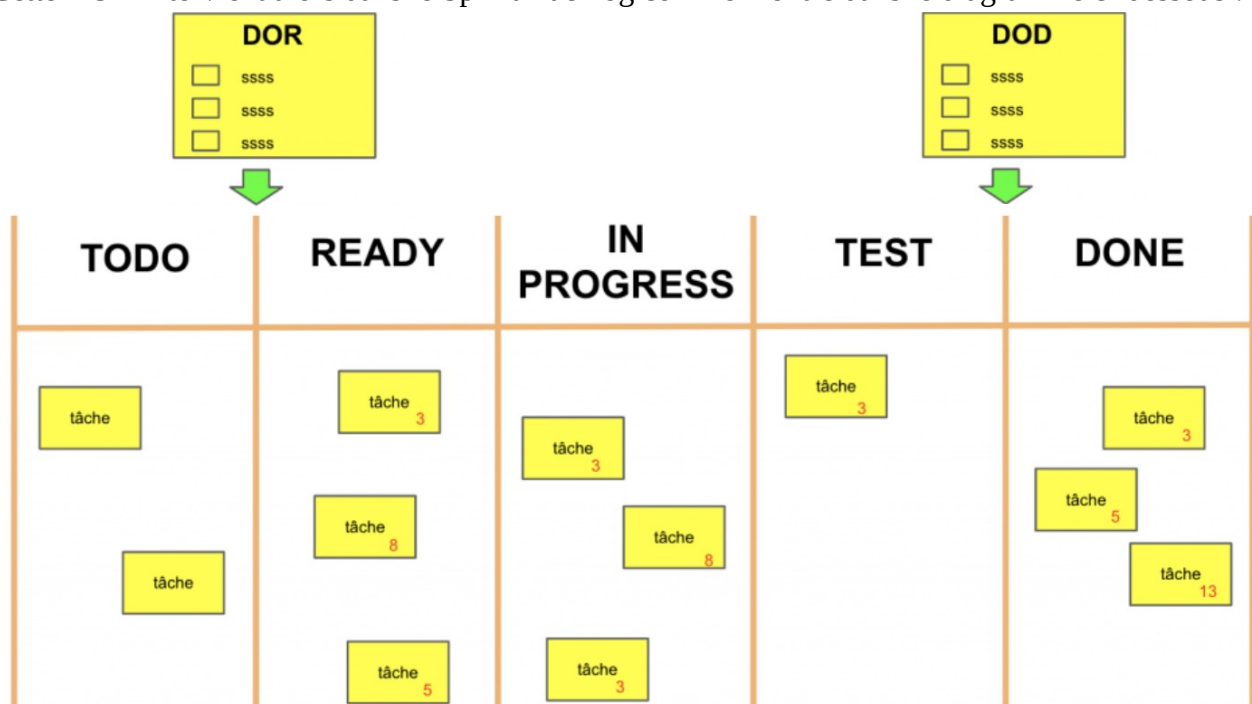
La DOD représente donc la liste des critères que l'équipe de développement va devoir s'imposer sur tous les éléments livrables durant le *sprint* : la qualité de la production d'un *sprint* est de la responsabilité de l'équipe de développement entière. C'est donc à elle de savoir qualifier, ou pas, si une US est prête à être livrée. Cette US, une fois livrée, répondra donc aux critères de qualité définis au sein de la DOD.

Exemple de DOD :

#### DOD (dev)

- ☐ Le développement est terminé
- ☐ Les TUs sont rédigés, réalisés, et passent au vert
- ☐ Les critères d'acceptations sont OK
- ☐ L'US est intégrable dans la démo de revue
- ☐ Les TUs et le code métier a été relu et refactoré
- ☐ Les initiales du développeur sont inscrites sur cette fiche

Cette DOD intervient alors dans le Sprint Backlog comme montré dans le diagramme ci-dessous :



### III.B.3. Epic

Une *epic* est une tâche fonctionnelle de haut niveau définie par le PO et intégrée au sein du *Product Backlog*.

Chaque *epic* sera décomposée par l'équipe Agile en une ou plusieurs US.

Un modèle d'*epic* pourrait être représenté par l'exemple ci-dessous :

- **Pour** <le client>
- **Qui** <fait une action>
- **La** <solution>
- **est** <quelquechose – le « comment »>
- **Qui** <fournit une valeur>
- **contrairement à** <un concurrent, une solution existante ou non>
- **notre solution** <fait quelquechose de mieux – le « pourquoi »>

### III.B.4. User Story

Une *User Story* (US) représente un besoin fonctionnel à obtenir au sein du produit.

La description fonctionnelle d'une US est de la responsabilité du PO et répond à un formalisme clair et précis : « *En tant que... Je veux que... Afin de...* », tel que présenté dans l'exemple ci-dessous lié à un site d'e-commerce :

- **EN TANT QU'**utilisateur
- **JE VEUX QUE** mon panier soit validé après l'avoir rempli
- **AFIN DE** conserver les articles que j'ai choisi.

Une US doit également répondre à différents critères constituant la DOR et indiquant si cette US peut être considérée comme prête à être développée.

## III.C. Les jalons et les cérémonies

### III.C.1. Sprint

Un *Sprint* est une phase temporelle séquentielle de développement du produit.

Le terme de « *Sprint* » est associé à des itérations de durées fixes (entre 2 et 4 semaines) pendant lesquelles l'équipe de développement réalise des tâches préalablement définies lors de la cérémonie du *Sprint Planning*.

Ces cycles permettent de décomposer un processus de développement souvent très complexe afin de le rendre plus simple et plus facile à réadapter, en fonction du résultat des évaluations intermédiaires.

Ainsi, un *Sprint* est en général organisé en quatre cérémonies (une cinquième peut être identifiée en fonction des besoins) ; ces cérémonies seront définies dans un paragraphe ci-dessous.

Le résultat d'un *Sprint* est appelé incrément, et correspond aux US déplacées dans la colonne *Fait* du *Sprint Backlog* durant le *Sprint*.

#### III.C.1.a. Préparation du *sprint*

La préparation du *sprint* se fait en présence de l'ensemble de l'équipe projet et du PO. La première action consiste à se mettre d'accord sur une stratégie opératoire et c'est à ce moment que débute réellement le *sprint*.

Durant cette réunion de préparation, l'équipe doit s'assurer que le PB est suffisamment fourni et détaillé. Ainsi, le PO se doit de structurer efficacement le PB et d'opérer une première sélection des fonctionnalités demandées. Ce premier travail de priorisation effectué, les objectifs à réaliser durant le *sprint* peuvent ensuite être discutés et décidés avec l'ensemble de l'équipe.

Afin de sélectionner les fonctionnalités présentes dans le SB qu'il faudra réaliser durant le *sprint*, chacune d'entre elles sera examinée et leur description affinée si besoin.

Pour mener à bien cette sélection, il est important que l'équipe ait conscience de sa **vélocité**, c'est-à-dire de la capacité de travail et de développement qu'elle pourra absorber durant le *sprint*. Cette connaissance est basée notamment sur l'expérience acquise durant les *sprints* précédents. Il est évident qu'elle sera plus difficile à évaluer au début d'un projet pour une équipe qui ne se connaît pas.

Ce n'est qu'au bout de plusieurs *sprints*, les membres de l'équipe ayant appris à se connaître et à travailler ensemble, qu'il sera alors possible de déterminer une vélocité moyenne stable et réaliste qui s'affinera au fur et à mesure des sprints, jusqu'à se stabiliser.

#### III.C.1.b. Périmètre et planification d'un *sprint*

Chaque *sprint* doit apporter des fonctionnalités supplémentaires à l'application en cours de développement qui doivent toutes être livrées lorsque le *sprint* en question se termine.

Comme énoncé dans le paragraphe précédent, il est de la responsabilité du PO de définir le périmètre du *sprint* et d'organiser le PB de façon à faciliter la construction du produit au fur et à mesure des *sprints*, en définissant des priorités de réalisation.

La décision d'inclure telle ou telle fonctionnalité dans le *sprint* courant est ensuite prise collectivement, durant la réunion de planification du *sprint*, appelée cérémonie du *Sprint Planning*.

De cette façon, la cible à atteindre pour la livraison à la fin du *sprint* est parfaitement définie.

Néanmoins, pour arriver à cette définition du périmètre du *sprint* courant, il est impératif que l'effort nécessaire à la réalisation de chaque fonctionnalité ait été estimé auparavant par l'équipe Agile.

*Nota : cet effort est quantifié et il sera alors possible de le comparer à la vélocité de l'équipe – notion décrite dans un paragraphe suivant.*

L'ensemble des fonctionnalités embarquées dans le *sprint* doit être cohérent pour ne pas obtenir simplement des fonctions déconnectées les unes des autres, mais bien une partie testable et évaluable de l'application finale.

Enfin, à l'issue du *sprint*, l'équipe projet doit être capable de présenter une démonstration des fonctionnalités développées, et de leur intégration dans les parties issues des *sprints* précédents, durant une cérémonie spécifique, appelée *Sprint review* et détaillée plus loin dans ce document.

### **III.C.1.c. Méthode de travail et déroulement d'un *sprint***

Durant la cérémonie du *Sprint Planning*, l'équipe projet va s'efforcer de choisir les US qui seront développés durant le *Sprint* en question, en sélectionnant les US déclarées « *ready* » dans les *sprints* précédents.

En effet, lors de cette cérémonie, il ne s'agira pas uniquement de piocher des éléments dans le PB pour constituer un *sprint*. Il sera alors nécessaire de définir une planification pour déterminer de quelle façon les objectifs fixés pourront être atteints.

La sélection et l'organisation des US à réaliser sont donc deux étapes clés de la méthode de travail en *sprints*. L'équipe de développement devra établir une véritable stratégie pour la réalisation technique des différentes fonctionnalités en se basant sur les estimations effectuées en amont.

Si le PO ne participe pas directement à l'élaboration de cette stratégie, il doit néanmoins être disponible à cette étape afin de répondre à toute question que pourrait encore se poser les développeurs.

A l'issue de l'étape de planification, l'équipe projet devra disposer d'une estimation réaliste de la quantité de travail nécessaire à la réussite du *sprint*. Les travaux pourront ainsi démarrer immédiatement et chacun des membres de l'équipe de développement aura une vision claire de l'effort à fournir jusqu'à la fin du *sprint*.

L'ensemble des fonctionnalités embarquées dans l'itération courante constitue le « *sprint backlog* ».

Durant tout le déroulement du *sprint*, le « *sprint burndown* » va permettre à tous de suivre l'évolution des différents travaux que se sont attribués individuellement chaque membre de l'équipe de développement. Ainsi, chacun pourra ainsi s'assurer que les différentes tâches sont terminées dans les temps.

Tout développeur dispose de son propre tableau d'avancement et le « *sprint burndown* » indique le travail qu'il reste à faire avant la fin de l'itération.

### III.C.1.d. Conclusion d'un *sprint*

A la fin d'un *sprint*, une réunion de démonstration est organisée, cérémonie appelée la *sprint review*.

A cette occasion, l'équipe devra présenter ce qui a été réalisé durant le *sprint* aux différentes parties prenantes.

Les experts métier et les utilisateurs finaux y sont naturellement conviés, et peuvent ainsi vérifier et valider que ce qui a été développé correspond bien à ce qui était demandé. Ils peuvent à ce moment-là faire différentes remarques et demandes de modification qui pourront être prises en compte ou non dans les prochains *sprints*, en fonction des décisions du PO et de l'équipe de développement.

Enfin, une dernière réunion a lieu en toute fin de *sprint* : la *rétrospective agile* ; cette cérémonie sera décrite plus loin dans ce document dans un autre paragraphe dédié.

**Dans le cadre de ce projet, un *sprint* aura une durée de 20 jours ouvrés, soit 4 semaines calendaires.**

### III.C.1.e. Le *Sprint0*, un *sprint* spécifique

Comme le titre de ce paragraphe l'indique, le *sprint0* est un *sprint* très très particulier.

Tout premier *sprint* du projet, aucune fonctionnalité ne sera livrée à son terme et l'équipe projet devra plutôt l'appréhender comme la construction d'un modèle sur lequel seront basés tous les *sprints* qui suivront.

Durant cette étape, l'ensemble des utilisateurs potentiels de la solution seront identifiés afin de collaborer avec eux pour dessiner le produit final. Les échanges entre le client et l'équipe projet seront donc particulièrement intenses durant le *sprint0* car ils vont permettre de définir ce que sera l'application attendue à la fin du dernier *sprint*, telles que l'ergonomie, les attentes en termes de sécurité, de fiabilité...

En outre, le *sprint0* servira aussi à définir la durée des sprints suivants, généralement de 2 à 4 semaines ; chaque *sprint* ayant la même durée.

De cette façon, il sera plus simple de mettre en place un rythme de travail, de faire adopter un certain nombre d'automatismes à l'ensemble de l'équipe projet, et de produire des indicateurs fiables pour le pilotage du projet lui-même.

En complément, les livraisons seront planifiées et le PB pourra être rempli, avec une première estimation et priorisation des fonctionnalités attendues.

Les environnements de développement, de tests, d'intégration et de livraison sont mis en place à ce stade de façon à être pleinement opérationnels dès le début du *sprint 1*.

Ainsi, même si le *sprint0* ne produit rien en matière de fonctionnalité, il représente la base sur laquelle vont s'appuyer l'intégralité des autres *sprints*, et, donc, le projet au final. Cette finalité met en avant la place spéciale du *sprint0* dans un projet agile : son importance est cruciale.

**Dans le cadre de ce projet, il est prévu que le *sprint0* ait une durée de 20 jours ouvrés, soit 4 semaines calendaires.**

### III.C.2. Cérémonies

Ce paragraphe a pour objectif de définir les différentes réunions appelée « *cérémonies* » ou « *rituels* » dans un cadre AGILE, et amenées à être potentiellement utilisées dans le cadre de la méthodologie et plus particulièrement pendant le déroulement d'un *sprint*.

#### III.C.2.a. Sprint planning

<b>Quand ?</b>	En tout début de <i>sprint</i>
<b>Pour qui ?</b>	Le PO et l'équipe de développement
<b>Durée ?</b>	De 30 minutes à 1 heure
<b>Objectifs ?</b>	<p>Le but de cette cérémonie est de définir le SB à partir du PB.</p> <p>Concrètement, le PO propose des US « <i>ready</i> » qu'il souhaite intégrer au <i>Sprint Backlog</i>.</p> <p>En général, le PO s'appuie donc sur le <i>Release Plan</i> pour choisir les <i>epics</i>/US.</p> <p>Pour cela, le PO doit prendre en compte la capacité de production de l'équipe de développement, son ressenti et sa capacité d'adaptation.</p> <p>A noter qu'à ce stade, le <i>Product Backlog</i> est censé contenir suffisamment d'<i>epics</i> et/ou d'US « <i>ready</i> » pour pouvoir définir un <i>Sprint Backlog</i>. Si ce n'est pas le cas, cette cérémonie ne peut se tenir et une cérémonie d'affinage, <i>sprint refinement</i>, du <i>Product Backlog</i> doit être effectuée en priorité.</p>

#### III.C.2.b. Sprint review

<b>Quand ?</b>	En fin de <i>sprint</i> , juste avant la <i>sprint retrospective</i> .
<b>Pour qui ?</b>	Toutes les parties prenantes du produit sont conviées à cette cérémonie : équipe projet, direction, actionnaires, utilisateurs finaux, etc...
<b>Durée ?</b>	De 30 minutes à 1 heure
<b>Objectifs ?</b>	<p>L'objectif principal de cette cérémonie est de permettre à l'équipe de développement de faire une démonstration de l'incrément créé pendant le <i>sprint</i> à l'ensemble des parties prenantes du produit.</p> <p>En général, cette cérémonie prend la forme d'une démonstration. Néanmoins, il est tout à fait envisageable de faire la présentation de l'incrément sur une branche de développement non finalisée.</p> <p>Cette cérémonie permet en outre d'exposer la capacité à produire et la production réelle de l'équipe de développement.</p>

### III.C.2.c. Sprint retrospective

Quand ?	En toute fin de <i>sprint</i>
Pour qui ?	L'équipe projet (PO, Scrum Master, équipe de développement)
Durée ?	De 1 heure à 2 heures
Objectifs ?	<p>Le but ici est de prendre en compte le ressenti des membres de l'équipe.</p> <p>Il n'y a pas forcément d'animateur désigné : l'ensemble des participants sont acteurs de cette cérémonie.</p> <p>La <i>sprint retrospective</i> n'est pas orientée sur la vision produit comme la <i>sprint review</i>. Cette cérémonie doit se focaliser sur le ressenti de l'équipe projet et donc d'un <b>point de vue humain</b>, tout en restant dans le cadre du sprint qui s'achève :</p> <ul style="list-style-type: none"><li>• « Qu'est-ce qui a bien fonctionné ? »</li><li>• « Qu'est-ce qui doit être amélioré ? »</li><li>• « Qu'est-ce qui est perfectible ? »</li><li>• « Qu'est-ce qui a failli ? »</li><li>• « Quelles sont les tensions qui ont pu émerger ? »</li><li>• etc...</li></ul> <p>Le but est de faire communiquer l'équipe. <b>L'ÉQUIPE SE DIT TOUT</b> : ce qui fait plaisir et également ce qui fâche - pas de langue de bois !!!</p> <p>L'équipe ne pourra s'améliorer que si elle identifie les points sur lesquelles elle doit focaliser ses efforts en matière d'amélioration et de points forts. Toutes les pistes seront à noter, mais pas forcément à prendre en compte dès le prochain <i>sprint</i>.</p> <p>En d'autres termes, cette cérémonie est l'occasion pour tous les membres de l'équipe agile de s'exprimer librement et d'expliquer ce qui s'est bien passé et ce qui s'est moins bien passé durant le <i>sprint</i>. Tous les domaines pourront être abordés : les conditions de travail, la façon de communiquer entre membre ou avec le client, les choix technologiques, les outils de travail...</p> <p>Cette réunion est la pierre angulaire de ce qui fait l'une des forces des méthodes agiles : l'<b>amélioration continue</b>. Des actions peuvent alors être décidées pour renforcer ce qui s'est bien passé, et pour corriger les éventuels problèmes rencontrés, afin que les prochains <i>sprints</i> soient encore plus efficaces et productifs.</p>



### III.C.2.d. Daily meeting

<b>Quand ?</b>	Tous les jours à heure fixe
<b>Pour qui ?</b>	Les membres de l'équipe de développement sont obligatoirement présents. Le Scrum Master et le PO peuvent y être invités si des précisions sont nécessaires.
<b>Durée ?</b>	Maximum 6 à 7 minutes pour une équipe de développement de trois personnes, soit environ deux minutes de temps de parole par membre de l'équipe de développement
<b>Objectifs ?</b>	Durant cette cérémonie, les membres de l'équipe de développement prennent tour à tour la parole afin de répondre aux trois questions suivantes : <ul style="list-style-type: none"><li>• « Qu'est-ce que j'ai fait depuis le dernier <i>daily meeting</i> ? »</li><li>• « Qu'est-ce que je compte faire d'ici le prochain <i>daily meeting</i> ? »</li><li>• « Quels sont les obstacles que j'ai rencontrés depuis le dernier <i>daily meeting</i> ? » C'est la réponse à cette question qui nécessite la présence du Scrum Master car il est de sa responsabilité d'enlever ces obstacles.</li></ul>

### III.C.2.e. Sprint refinement

<b>Quand ?</b>	A la discrétion du PO et de l'équipe de développement
<b>Pour qui ?</b>	Le PO et l'équipe de développement
<b>Durée ?</b>	A la discrétion du PO et de l'équipe de développement
<b>Objectifs ?</b>	<p>Cette cérémonie est facultative et se tient exclusivement sur demande du PO ou de l'équipe de développement.</p> <p>Cette cérémonie a pour objectif de faire évoluer les <i>epics</i> et/ou les US du <i>Product Backlog</i> de façon à pouvoir les qualifier de l'état « <i>ready</i> ».</p> <p>La finalité de cette cérémonie est de faire en sorte que chaque critère de la DOR vis à vis d'une US soit rempli dans l'optique de pouvoir l'intégrer à un <i>sprint</i> lors de la cérémonie d'un futur <i>Sprint Planning</i>.</p>

## III.D. Outils organisationnels

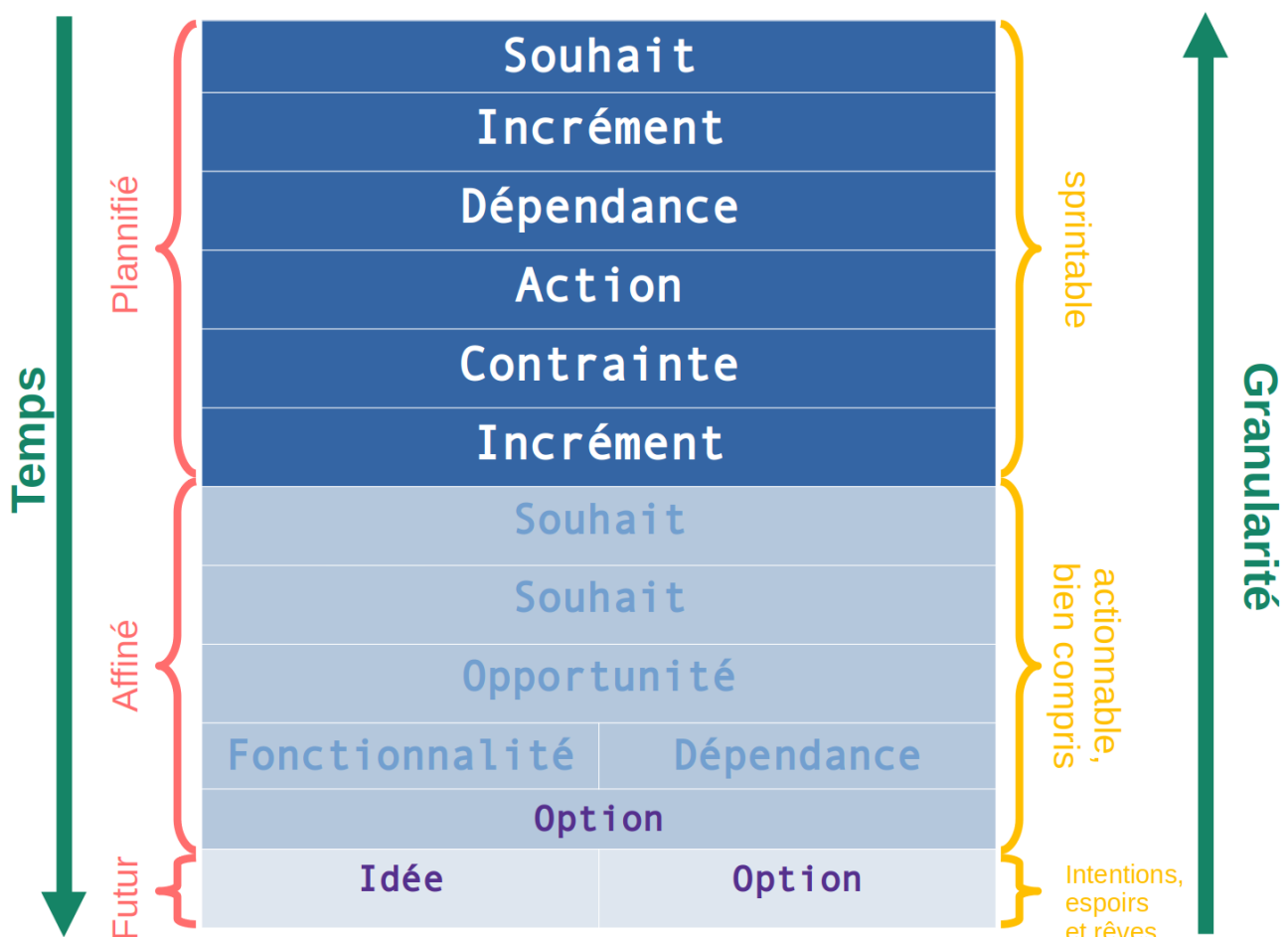
### III.D.1. Product Backlog

Le *Product Backlog* (PB), dont la **gestion est de la responsabilité exclusive du PO**, représente la vision du produit telle que se la représente le client final : c'est donc la liste des besoins fonctionnels de haut niveau recueillis pour créer le produit fini.

Le PB est ainsi composé de l'ensemble des *epics* définies par le PO, et est la source à partir de laquelle le *Sprint Backlog* est alimenté lors de la cérémonie du *Sprint Planning*.

Le contenu du PB est travaillé sur demande du PO lors de la cérémonie d'affinage du PB, dont l'objectif est de répondre aux différents critères de la DOR afin d'aboutir à des US prêtes à être développées (US « ready »).

Par exemple, un PB peut se représenter comme une pile de tâches fonctionnelles de haut niveau (*epic*) comme présentée dans le diagramme ci-dessous :



### III.D.2. *Sprint Backlog*

Le *Sprint Backlog* (SB) est construit lors de la cérémonie de *Sprint Planning* à partir du *Product Backlog* et permet d'indiquer à l'équipe de développement la vision de ce qu'elle aura à produire durant le *sprint*, au travers de ce carnet de route.

Le *Sprint Backlog* est ainsi composé d'US présentent dans le PB et, chose importante, à l'état « *ready* ».

Le *Sprint Backlog* sert donc à définir l'objectif du *sprint* d'un point de vue fonctionnel.

Dès lors que le *Sprint Backlog* est défini, il se formalise sous la forme d'un tableau où chaque colonne représente un état, tel que montré dans l'exemple suivant :

A Faire	En cours	Examen	Fait
US_2	US_3		US_5
US_4	US_1	US_6	US_8
US_9			

Dans l'exemple de *Sprint Backlog* ci-dessus , les colonnes représentent :

- **A Faire** : les US à produire avant la fin du sprint ;
- **En cours** : les US en cours de réalisation ;
- **Examen** : les US à relire pour être sûr qu'elles ont produit l'effet escompté ;
- **Fait** : les US terminées répondant à la DOD.

### III.D.3. Release Plan

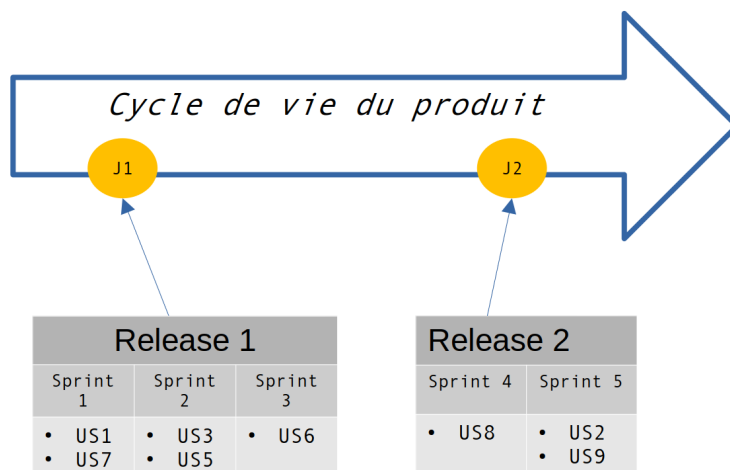
Le *Release Plan* (RP) définit et planifie les versions d'un produit, sachant qu'une version du produit est le résultat d'un ou plusieurs *sprints*.

Pas toujours apprécié dans le monde de l'agilité, le contexte projet impose parfois de devoir réaliser un *Release Plan* ; en général, cet artefact est imposé pour dans de nombreux projets dans lesquels des jalons sont imposés par le client lui-même.

La mise en place d'un *Release Plan* nécessite cependant certains pré-requis :

- les objectifs sont connus tant en termes de dates que de fonctionnalités à intégrer dans les différentes *releases* ;
- le *Product Backlog* doit être composé uniquement d'US à l'état « *ready* » (incluant l'estimation des US) ;
- la durée des *sprints* est fixée dans l'optique de pouvoir estimer la capacité de l'équipe de développement à produire ;
- la stabilité de l'équipe doit être assurée (pas de perturbation hors projet).

Un *Release Plan* peut être schématisé assez facilement de la manière suivante :



Avec par exemple :

- J1 = 26/04/2024
- J2 = 06/08/2024

### III.D.4. Vitesse et point d'US

La vitesse est une métrique clé en Agile correspondant à la vitesse moyenne de production de l'équipe Agile. Ainsi, la détermination de la vitesse de *sprint* de l'équipe Agile permet de comprendre quand atteindre des jalons et, donc, d'anticiper les délais relatifs à la ligne d'arrivée.

L'utilisation de la vitesse peut également aider à réduire les risques de promesses excessives lors de l'acceptation des livrables du client. La vitesse agile est un outil à utiliser en interne, et pas nécessairement à utiliser pour des déterminations externes. L'équipe agile peut alors l'utiliser pour suivre les estimations et les réalisations sur plusieurs *sprints*, ce qui lui donne une idée précise de la vitesse à laquelle elle peut appréhender un *backlog*. Cela l'aide à examiner ses progrès et ses points forts et à apprendre comment améliorer ses métriques.

### III.D.4.a. Calcul

Avant de commencer à calculer la vélocité de l'équipe agile, il est nécessaire d'effectuer au moins trois à cinq *sprints*. Ce délai permet alors à une équipe novice en matière de gestion de projet Agile de s'habituer au flux de travail et à tout changement que l'équipe subit.

La vélocité fluctuera au cours des *sprints* initiaux et ne se stabilisera qu'après trois *sprints* ou plus.

Pour travailler sur la formule, il est nécessaire de connaître la valeur en points de chaque US ; cette métrique d'US est une mesure du point d'histoire relative. Elle peut être calculée de différentes manières pour différentes organisations.

**La règle d'or pour déterminer un point d'histoire est de trouver l'histoire (US) la plus simple, de lui attribuer un point, puis de l'utiliser pour évaluer le reste.**

Il est alors possible d'opter pour deux échelles afin de déterminer ces points d'histoire : une **échelle linéaire** ou une **séquence de Fibonacci**.

En espaçant suffisamment les points de l'histoire (ou point d'US), l'équipe agile pourra se faire une meilleure idée de l'importance des différentes tâches lorsqu'elle les passera toutes en revue.

Pour calculer la vélocité d'un *sprint*, il faut connaître :

- combien d'US l'équipe doit-elle terminer durant un *sprint* ;
- le nombre de points d'histoire que vaut une US.

Ensuite, l'équipe agile devra regarder le nombre d'US terminées et additionner les points.

A titre d'exemple, cette étude va suivre un cas d'usage hypothétique :

- *Sprint 1* : l'équipe projet s'est engagée sur 8 US, et chaque US équivaut à 3 points d'histoire. Admettons que l'équipe ne termine que 4 US à l'issue du *sprint*. Elle obtiendra alors 12 points d'US.
- *Sprint 2* : l'équipe projet s'est engagée sur 10 US, et chaque US équivaut à 5 points d'histoire. Admettons que l'équipe ne termine que 7 US à l'issue du *sprint*. Elle obtiendra alors 35 points d'US.
- *Sprint 3* : l'équipe projet s'engage sur 9 US, et chaque US équivaut à 4 points d'histoire. Admettons que l'équipe ne termine que 7 US à l'issue du *sprint*. Elle obtiendra alors 28 points d'US.

Maintenant que les vitesses des 3 premiers *sprints* sont connues, il est possible d'en calculer la moyenne et donc d'obtenir la vélocité de l'équipe sur ces 3 sprints :

$$\text{Vitesse moyenne de sprint} = \text{vélocité} = (12+35+28)/3 = 56$$

### III.D.5. Poker Planning

Dans le précédent paragraphe, il a été question de points d'histoire (ou point d'US). Ces derniers servent à évaluer l'énergie qu'une équipe projet doit investir pour réaliser une US.

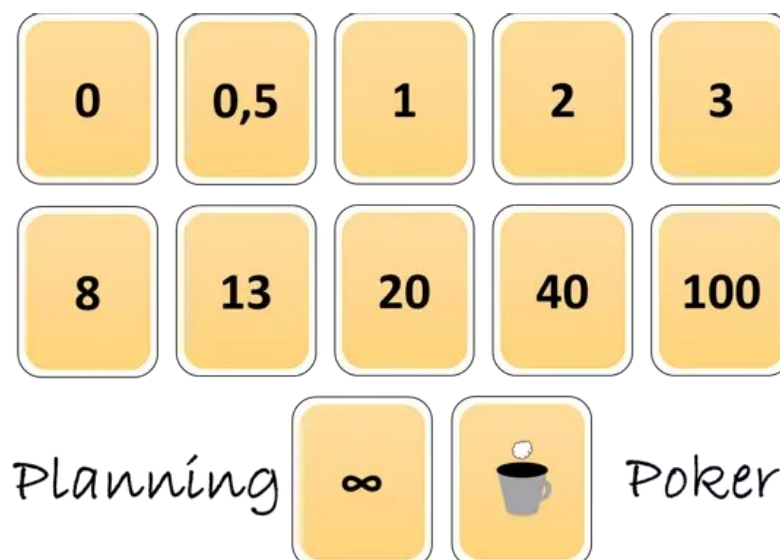
Pour calculer ces points d'US, l'équipe agile peut faire appel à un jeu d'estimation appelé *poker planning*.

Le planning poker, également connu sous le nom de *Scrum poker*, est une technique d'estimation ludique basée sur le consensus, principalement utilisée pour estimer l'effort ou la taille relative des objectifs de développement dans le développement de logiciels. Le *poker planning* sert ainsi à estimer l'énergie, ou points d'US, à investir pour la réalisation de chaque US.

Ainsi, pour chaque US à estimer, le jeu est composé de 4 étapes consécutives :

- **Etape 1** : le PO lit l'US à estimer ou décrit une fonctionnalité l'équipe de développement.
- **Etape 2** : Chaque membre de l'équipe fait des estimations en jouant des cartes numérotées face cachée sur la table, sans révéler leur estimation (ces cartes comportent en général les valeurs de la suite de Fibonacci : 1,2,3,5,8,13,20,40).
- **Etape 3** : les cartes sont montrées et affichées simultanément.
- **Etape 4** : Les estimations sont ensuite discutées et les estimations hautes et basses sont expliquées. L'équipe discute entre elle pour trouver un consensus d'estimation, c'est à dire une valeur pour laquelle l'équipe agile est d'accord majoritairement.

Dans les points ci-dessus, les étapes 3 et 4 sont à répéter autant de fois que nécessaire jusqu'à ce que l'équipe aboutisse à une valeur de points d'énergie commune.



### III.D.6. Graphique de *sprint burndown*

Un graphique de *sprint burndown* est un outil permettant de mesurer visuellement la quantité de travail accompli en une journée par rapport au taux d'achèvement prévu pour le *sprint* ou la version existante.

Puisqu'il montre les progrès quotidiens, l'équipe peut évaluer si elle est en mesure d'atteindre ses jalons et de fournir la solution requise dans le temps imparti.

Ainsi, un graphique de *sprint burndown* révèle plusieurs indications à l'équipe projet :

- L'**estimation totale** : il s'agit de l'effort total en heures de travail que l'équipe s'est engagée à effectuer. Cela inclut les US ou la clôture des tickets ou des problèmes.
- La **quantité de travail restant à effectuer** : le graphique de *sprint burndown* indique non seulement la quantité de travail achevé, mais également le travail total restant à effectuer.
- Le **nombre total de jours ouvrés** : cela représente le nombre total de jours ouvrés dans un *sprint* ou la durée du *sprint*. Cela doit être indiqué dans le tableau d'avancement, car l'équipe devra calculer la quantité de travail qui reste à accomplir et le temps qu'elle peut consacrer à l'élément engagé.
- L'**effort idéal** : l'effort idéal est un critère par rapport auquel l'équipe peut mesurer sa performance. Il est représenté en calculant la quantité exacte d'effort restant qui doit être brûlé.
- L'**effort réel** : il s'agit de la somme totale de tous les efforts restants à la fin de chaque journée.



### III.D.7. Graphique de *sprint burnup*

Un graphique de *sprint burnup* est relativement similaire à un graphique de *sprint burndown*.

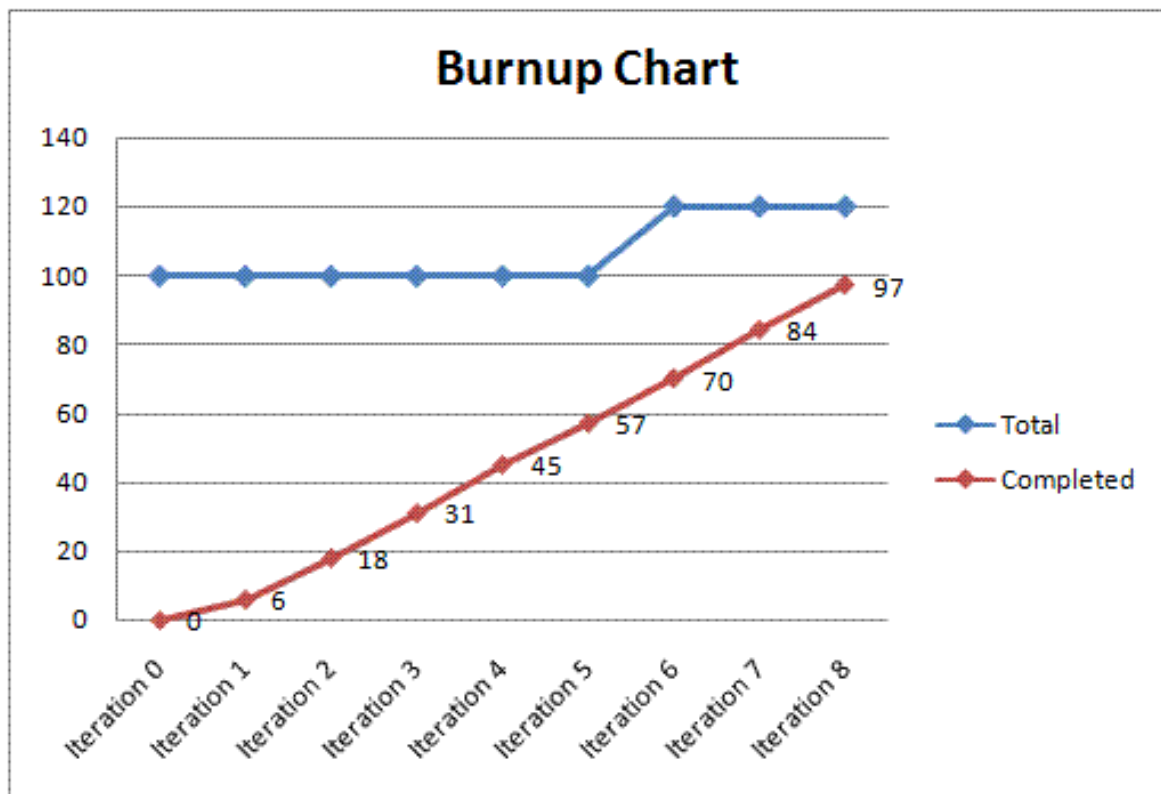
La principale différence est qu'il suit le travail terminé plutôt que le travail restant.

Ce graphique montre la quantité de travail achevée par rapport à la portée totale du *sprint*.

Une autre différence clé est que le *sprint burnup* est plus efficace pour représenter le fluage (déformation) de la portée.

Bien que cette déformation de la portée soit visible sur un *sprint burndown*, celle-ci est d'autant plus claire et explicite sur un *sprint burnup*.

En effet, ce type de diagramme comprend une ligne distincte reflétant le périmètre total ; si, ne serait-ce que 5 petits points d'US sont ajoutés à la portée globale, cela ressort irrémédiablement sur un *sprint burnup*.



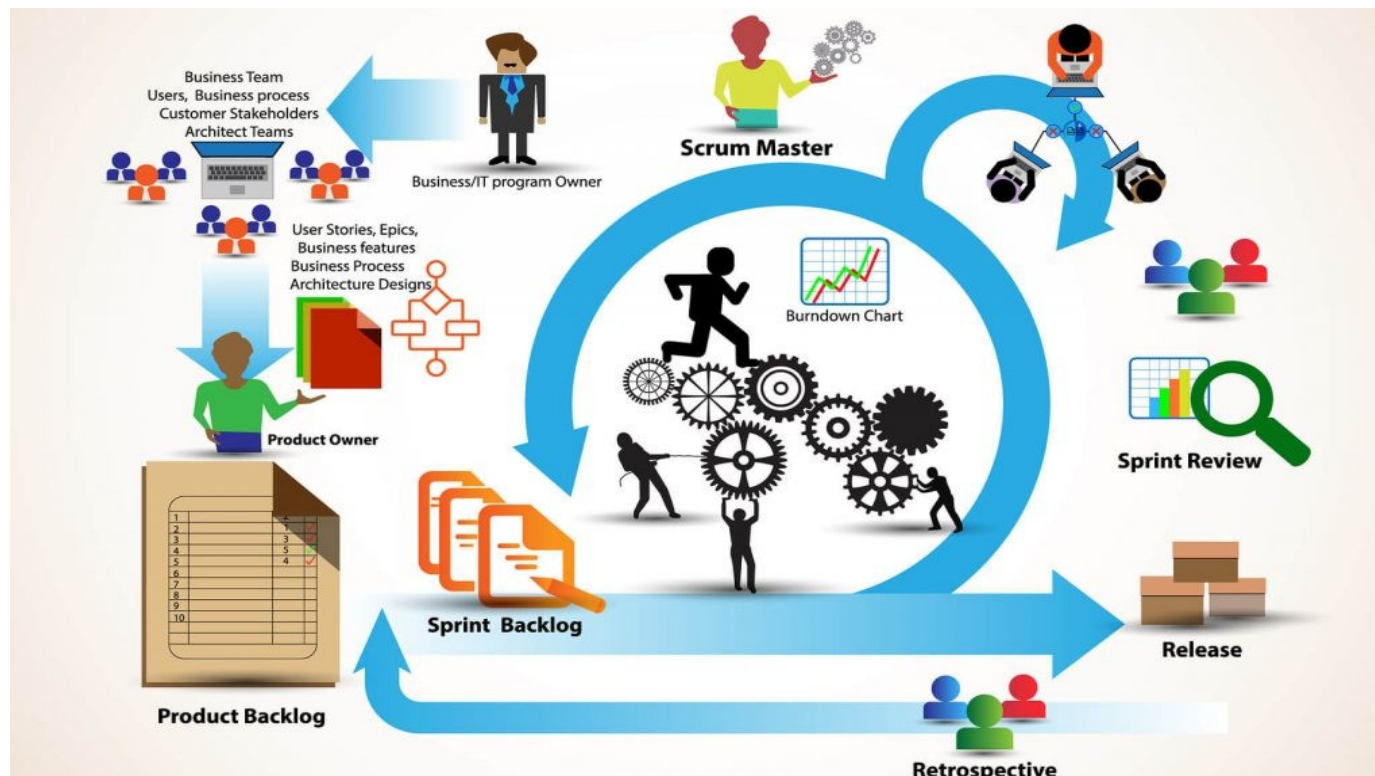


## IV. synthèse

Dans les paragraphes précédents, il a été question de :

- **acteurs** : *Product Owner, Scrum Master, Développeur* ;
- **artefacts** : *Epic, User Story, Definition Of Ready, Definition of Done* ;
- **cérémonies** : *Sprint Planning, Daily Meeting, Sprint Refinement, Sprint Review, Sprint Retrospective* ;
- **outils d'organisation** : *Product Backlog, Sprint Backlog, Release Plan, vélocité, poker planning, points d'US.*

Tous les éléments référencés dans les points ci-dessus sont orchestrés au sein d'une seule et même partition, nommée **sprint**, structurée et organisée selon le schéma ci-dessous :





GITMEMORY